Interactive Program Debugging

Robert F. Erbacher
Department of Computer Science, LI67A
University at Albany - SUNY
Albany, NY 12222
erbacher@cs.albany.edu
Phone: 518-437-3758
Fax: 518-442-5638

Abstract

While debugging technology has advanced in recent years, particularly through the incorporation of graphical user interfaces, there is still a need for new advances to improve the efficiency of the process. As software applications continue to become ever more complex this need is continuing to increase. This work is a follow on to the visualization work that has been ongoing for a number of years and applies steering metaphors to the debugging process to address the inefficiency of those parts of the debugging cycle in which a developer has identified an error and must identify and incorporate a verified correction. In this way, the developer can change the code dynamically during run-time without the need to leave the debugging environment until the correct modifications have been determined, greatly improving the efficiency of the debugging process.

Keywords: Visualization, Debugging, Interactive Computational Steering

1. Introduction

Recent software development tools have been incorporating a greater extent of visual capabilities than ever before. Graphical user interfaces are common place in debuggers and visual languages. Many of these ideas are derived from previous research projects and are geared towards developing environments that are easier to use. This derives from the fact that most debuggers are not used or are not used adequately due to their high learning curve. By incorporating a greater visual paradigm, making the use of such debugging environments easier, the environments receive greater use and thereby increase the productivity of the developer by improving the efficiency of the debugging cycle. For example, in the textual version of most debuggers the user would have to type a command such as “break 26” to halt execution on line 26. In a visual paradigm the user must merely locate the correct line and click in the left side of the window to turn on a break point, most other features have similar visual characteristics. These visual environments also allow the user to view activity in multiple windows (views) simultaneously. In this way, the user can be examining the code in one window and selected data values and attributes in a second window. Past research has applied visualization to deal with the need to represent large amounts of data under the debugger control [1, 2, 3, 4, 5, 6], since clearly textual representations will be inadequate.

However, these tools and advances are still greatly limited and do not reduce some of the most time consuming aspects of the debugging process, namely error identification and error correction. Our recent work has been geared towards applying visual steering to greatly improve these aspects of

the debugging process [7]. In a typical environment, even one that provides extensive visual capabilities, the user is required to return to the original source code to make modifications and "hopefully" correct the error. Unfortunately, most of the time it takes several iterations before the correct modifications are determined that will completely eradicate the error within the code.

This aspect of the debugging process consumes an enormous amount of time during which the developer is being primarily unproductive. The steps involved generally require that the source code be modified with the proposed change, the source code be saved, the source code be recompiled, the application be re-executed, and the program be re-run to the point of the fault. If you consider a large software project, each recompilation of the environment can take several minutes. Execution of the environment, to get to the location of the previously identified error, can also take several minutes, perhaps longer depending on the nature of the environment. Some simulations can take hours or days to get to a particular point. Even in shorter run environments the user must reload the correct data and specify parameters correctly. This process alone is prone to human error.

This paper describes our research into visual steering of the debugging process. In particular, we are looking at methods for allowing the user to dynamically change the programs behavior, i.e., changing aspects of the source code, during run-time through steering. This capability combined with other visualization and steering capabilities begins to provide an environment that truly has the potential to reduce the time consumed by the debugging process.

2. Previous Work

The visual representation of data on both serial (Balsa [8]) and parallel and distributed systems (Maritxu [1], IVE [9], Prism [2], VisCon [3, 4, 5, 7]) has been given substantial attention. Visual representation of data in debugging, performance tuning, and application comprehension tasks has become common. However, the usefulness of the techniques has been hindered by the limited application of interaction and steering and the limited scope of the data that is represented. Parallel and distributed systems, in particular, have received an enormous amount of effort directed at providing additional tools to aid in their analysis, a consequence of their complexity.

Interactive computational steering [10] is a technique through which the user can change parameters on the fly to help direct the program execution to more interesting results and to gain greater understanding of the data [11]. Steering has been applied to many areas, both serial and parallel. Falcon provides steering of large-scale parallel programs to improve application performance and affects the application's execution behavior [12]. Progress is a toolkit for providing steering in any application through instrumentation, primarily geared towards steering long running parallel programs [13]. SCIRun is design to aid comprehension of cause-effect relationships in large scale computations [14, 15]. CUMULVS provides a general interactive steering environment through the implementation of a check pointing mechanism that provides fault tolerance and speeds error identification and correction [16, 17]. These environments have found many benefits from steerable environments, including:

- Greater understanding of the data and the effect of various parameters
- Improved application performance by directing execution around branches identifiable as not leading towards a viable solution
- Aided detection of errors
- Aided comprehension of the algorithm and the theoretical basis of the algorithm
- Improved performance through manual load balancing

All of these environments require that the user manually instrument the program, identifying code or data elements that the programmer believes will be desirable to be modified during execution. This limited use of instrumentation is due primarily to concerns of possible performance degradation and the need to maintain high rates of execution but adds another level of human error and time consumption.

3. The VisCon Environment

The VisCon environment is designed to provide an interactive computational steering environment for program comprehension and debugging. VisCon provides a fine grain instrumentation environment suitable for automatic instrumentation. The environment consequently incurs a greater performance hit than seen in the previously discussed steering environments. However, this fine grain instrumentation provides instant access to all variables in the application, a necessity for true debugging when it is not known where an error may occur within the source code or which variables will contribute to the error. The environment is not geared towards performance tuning and thus the performance should not be an issue. There is of course the potential for substantial impact due to the probe effect. Errors susceptible to probe type effects will require additional capabilities to detect and correct.

An important goal for our work is to provide techniques that will allow the user to interact directly with the executing program and make changes without having to restart the program. Our desire was to provide capabilities to aid the user in gaining an in-depth understanding of the program in a timely fashion without having to waste time editing, compiling, and rerunning the program. To this end, we explored the applicability of interactive computational steering to the debugging of programs.

4. Dynamic Instruction Modification

Our steering paradigm allows the modification of any instruction or command within the source code during the execution of the application. The steering may be performed both at the level of individual instructions and at the equation level; i.e., modifying an assignment operation. In locations where user changes to an instruction occurs, the environment provides visual feedback to indicate when those modified instructions are being executed. This is indicated through the incorporation of an indicator within the status bar, similar to a LED indicator.

To modify an instruction, the user will pause the execution of the application, provided by VCR like controls incorporated into the steering environment, and bring up the instruction editor. The user types in the new operation or equation that is then stored in a list associated with the original instruction (Figure 1). An integrated interpreter and parser evaluate this new equation whenever necessary.

The user can change any instruction within the program. Namely, the user can change not only a single operation within an equation but also the entire equation with a single replacement equation. If the user modifies an instruction more than once then the last modification takes precedence and provides the result applied. This technique can be quite useful in controlling applications. It allows modification to a routine dynamically without recompiling or even leaving the executing environment, greatly improving the debugging process. Once an error is identified and located using the visualization tools the instruction steering techniques can be used to apply a proposed change. Execution continues from that location, with no need to exit the environment, recompile, or attempt to relocate the error. The user can immediately validate the proposed change. Should the change not be sufficient then further modifications can be applied. This allows the user to completely verify the correctness of a proposed solution before returning to the actual source code, greatly reducing the time requirements of the debugging cycle.

5. Instrumentation for Instruction Level Steering

The following code segment shows how instrumentation instructions within a program can allow the modification of the algorithm. The equation to solve for a variable, res, is res=a*b+c+1. We add instrumentation instructions for each operation (*, +, +, and =). The user can modify a single operation; i.e., the user can change the multiplication to be division or the multiplication to be a whole equation in and of itself. Alternatively, the user can modify the assignment operation to change the entire equation

```
Func(int a, b, c)
{
  int res;

  res=(opres=a*b, opres=instr(ID, MULT, a, b, opres) +c,
       opres=instr(ID+1, ADD, internal, c, opres)+1,
    opres=instr(ID+2, ADD, internal, 1, opres)),
       opres=instr(ID+3, ASSIGN, internal, internal, opres);
  printf("%d\n", res);
}


int instr(int ID, int operator, int operand1, int operand2, int
result)
{
  if (checkModified(ID))
     internal=modifiedResults(ID, operand1, operand2);
  else
     internal=result;

  return internal;
}
```

Figure 1: Example source code showing code instrumentation paradigm.

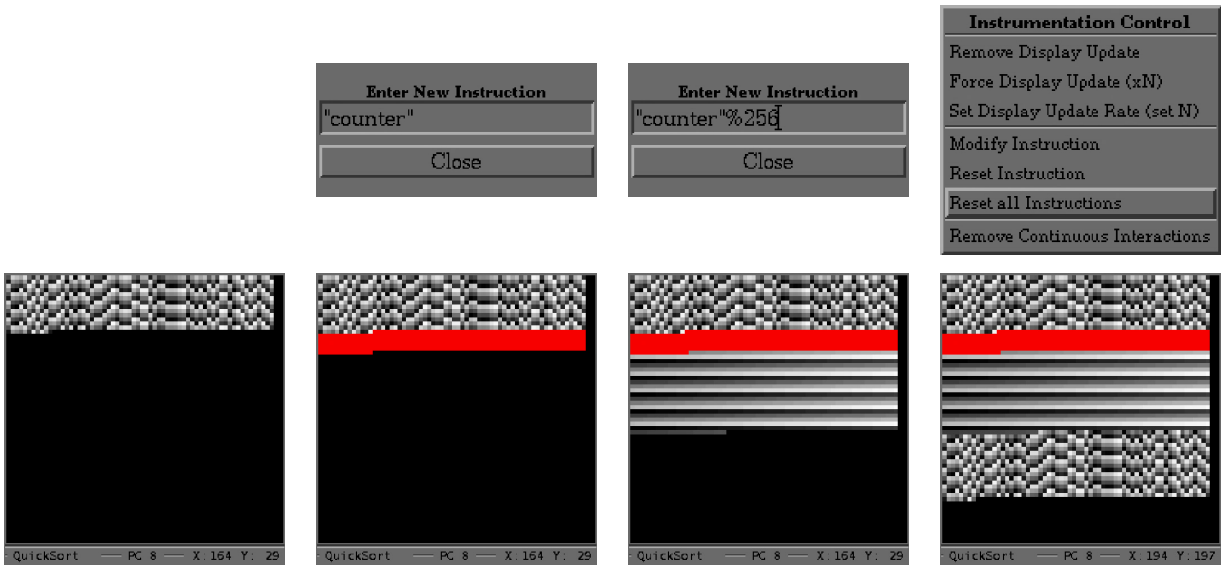
as a whole in a single swoop. If you look at the sample code (Figure 1) for the instrumentation instruction itself you can see that all the instrumentation instruction need do is check a database to see if the user has applied a modification to the instruction currently under investigation. If the user has applied a modification then the application uses the results of the modified instruction for the remainder of the computation rather than the original results. Otherwise, the application uses the original results. The use of the comma operator to add instrumentation instructions ensures that the remainder of the equation will use the value returned by the instrumentation instructions correctly. This allows the instrumentation instruction to control completely the value used by the remainder of the equation.

6. Example

Figure 2 shows an example of instruction modification in action. Figure 2a shows random values being generated by a sample algorithm with a simple visual representation of the data values being generated. Figure 2b shows that the generation of values has been modified to be equal to a variable within the code called “counter”, which is in fact the loop counter. All variable instantiations are themselves instrumented. This allows the environment to provide immediate access to all available variables through the graphical user interface. When the scope of a variable is reached it is subsequently deleted from the list of variables.

Since the loop counter generates values outside our expected range of 0..255 the elements are highlighted in red. In the environment, the range of expected values is configurable, providing an additional level of verification within the visualization display. Figure 2c shows the refinement process involved in modifying instructions. We modify our instruction to modulos the “counter” variable by 256. This gives us values back in our expected range of 0..255. Figure 2d shows the removal of the modified instructions. Ultimately, any set of expressions could be integrated into the command modifications.

A more subtantive example can be considered with a quick sort routine. The steering routines will allow the user to experiment with different equations for generating the partition element. Thus, given typical data for an application an



(a) (b) (c) (d)

Figure 2: Sample application of instruction level steering.

optimal partition element algorithm can be selected in a timely fashion. There is no need to recompile the code. The different algorithms can be entered and compared for performance considerations. The most effective algorithm can then be incorporated into the actual source code. Such a paradigm is much more efficient than the typical paradigm that would require that the source code be modified, recompiled, and re-executed. In fact, viewing the execution of one particular algorithm may lead to the derivation of an effective algorithm based on the viewed behavior. This is a common goal of steering environments to add another level of efficiency.

7. Conclusions

Steering can be applied at many levels in the software development process. We have developed novel steering capabilities to dynamically allow the modification of instructions during execution. This helps the user to identify the correct solution to an identified error and greatly improves the efficiency of the debugging cycle by eliminating much of the tedious and unfruitful recompilation and re-execution of the application. This improved efficiency will have the added benefit of maintaining the users focus on the task at hand.

The techniques we are developing will apply to a wide variety of environments. In particular, the steering metaphors will apply across system architectures and will apply to serial as well as parallel and distributed systems. By providing such consistent interfaces across architectures the environments become much more familiar and easier to use. Parallel and distributed systems will need additional capabilities to deal with their unique characteristics but the majority of the process will essentially remain identical.

References

[1] Eugenio Zabala and Richard Taylor, “Maritxu: Generic Visualisation of Highly Parallel Processing,” Programming Environments for Parallel Computing; Proceedings of the IFIP WG 10.3 Workshop on Programming Environments for Parallel Computing, Elsevier Science Publishers B.V., 1992, pp. 171-180.

[2] Steve Sistare, Don Allen, Rich Bowker, Karen Jourdenais, Josh Simons, and Rich Title, “Data Visualization and Performance Analysis in the Prism Programming Environment,” Proceedings of the IFIP WG 10.3 Workshop on Programming Environments for Parallel Computing,

Elsevier Science Publishers B.V., 1992, pp. 37-52.

[3] Robert F. Erbacher, "Visual debugging of data and operations for concurrent programs", Proceedings of the SPIE '97 Conference on Visual Data Ecploration and Analysis IV, San Jose, CA, February, 1997, pp. 120-127.

[4] Robert F. Erbacher and Georges G. Grinstein, "Program visualization: bringing visual analysis to code development," Proceedings of the SPIE '99 Conference on Visual Data Exploration and Analysis VI, San Jose, CA, January, 1999, pp. 32-39.

[5] Robert F. Erbacher and Georges G. Grinstein, "Visualization of Data for the Debugging of Concurrent Systems," Proceedings of the SPIE '96 Conference on Visual Data Exploration and Analysis III, San Jose, CA, February, 1996, pp. 140-149.

[6] C.S. Collberg, S. Davey, T.A. Proebsting, "Language-agnostic program rendering for presentation, debugging and visualization," *Proceeding 2000 IEEE International Symposium on Visual Languages*, Seattle, WA, September, 2000, pp. 183-190.

[7] Robert F. Erbacher, "Visual Steering for Program Debugging," Proceedings of the SPIE '2000 Conference on Visual Data Exploration and Analysis VII, San Jose, CA, January, 2000, pp. 106-113.

[8] M. H. Brown and R. Sedgewick, "Techniques for Algorithm Animation," IEEE Software, Vol. 2, No. 1, 1985, pp. 28-39.

[9] Mark Friedell, Sandeep Kochhar, Mark LaPolla, and Joe Marks, "Integrated Software, Process, Algorithm and Application Visualization," Journal of Visualization and Computer Animation, 1992.

[10] Weiming Gu, Jeffrey Vetter, and Karsten Schwan, "An Annotated Bibliography of Interactive Program Steering," Georgia Institute of Technology College of Computing Technical Report GIT-CC-94-15, 1994.

[11] Robert van Liere, Jurriaan Mulder and Jack van Wijk., "Computational Steering," Future Generation Computer Systems, Vol. 12, nr. 5, 1997.

[12] Weiming Gu, Greg Eisenhauer, Eileen Kraemer, Karsten Schwan, John Stasko, Jeffrey Vetter, "Falcon: On-line Monitoring and Steering of Large-Scale Parallel Programs," Frontiers 95, McLean, Virginia, February, 1995.

[13] Jeffrey Vetter and Karsten Schwan, "Progress: a Toolkit for Interactive Program Steering," Proceedings of the International Conference on Parallel Processing 1995, Oconomowoc, Wisconsin. August, 1995.

[14] Jeffrey Vetter and Karsten Schwan, "Techniques for High-Performance Computational Steering," IEEE Concurrency, pp. 63-74, Vol7. No. 4, 1999.

[15] S.G. Parker, M. Miller, C.D. Hansen, and C.R. Johnson, "An integrated problem solving environment: The SCIRun computational steering system," Proceedings of the 31st Hawaii International Conference on System Sciences (HICSS-31), volume VII, pp. 147-156, Jan. 1998.

[16] J. A. Kohl, "Interacting with High-Performance Scientific Simulations Using CUMULVS: Visualization, Computational Steering, and Fault Tolerance," Graduate Colloquium, Department of Electrical and Computer Engineering / CSE, University of Louisville, Louisville, KY, October 7, 1999.

[17] J. A. Kohl, P. M. Papadopoulos, "Efficient and Flexible Fault Tolerance and Migration of Scientific Simulations Using CUMULVS," Proceedings of the 2nd SIGMETRICS Symposium on Parallel and Distributed Tools, Welches, OR, August 1998.